สุทธิศักดิ์ พงศ์ธนาพาณิช

เปิดโลกการเข้ารหัสข้อมูลด้วยภาษาซี ตอน การบีบอัดข้อมูลด้วยวิธี Sliding Window #1

การบีบอัดข้อมูลด้วยอัลกอริธึม LZSS ซึ่งเป็นอัลกอริธึมที่ได้รับการพัฒนาต่อเนื่องมาจากอัลกอริธึม LZ77

ในบทความตอนที่ผ่านมา ผู้เขียนได้เปิดแนวความคิดใหม่เกี่ยวกับการบีบอัดข้อมูล โดยได้เสนอแนวคิดของการบีบอัดแบบ dictionary-based ซึ่งมีพื้นฐานที่แตกต่างจากอัลกอริธึมของ huffman หรือ shannon-fano อย่างสิ้นเชิง โดยผู้เขียนได้จบบทความในตอนที่ผ่านมาด้วยอัลกอริธึมของ LZ77 ซึ่งได้มีการนำเอาเทคนิค sliding window มาใช้ในการเข้ารหัสสายข้อมูล โดยการสมมติว่าสายข้อมูลที่จะถูกเข้ารหัสจะถูกนำมาเก็บไว้ในบัฟเฟอร์ ซึ่งมีขนาดตามที่ต้องการ เช่น 2 กิโลไบต์ เป็นต้น จากนั้นก็จะทำการเปรียบเทียบสตริงในบัฟเฟอร์เพื่อทำการเขารหัส จากนั้นก็จะมีการเลื่อนสตริงตั้งแต่ตัวแรกจนถึงสตริงที่ถูกเข้ารหัสออกจากบัฟเฟอร์ และมีการอ่านสตริงขนาดเท่ากันเข้ามาแทนที่ในบัฟเฟอร์ เหมือนกับว่าเราได้มีการเลื่อนหน้าต่างขนาด 2 กิโลไบต์ ไปตามสายสตริง เพื่อทำการเข้ารหัสสตริงนั่นเอง ซึ่งเป็นที่มาของคำว่า sliding window ตามที่กล่าวมาแล้วในบทความที่ผ่านมา

สำหรับในบทความตอนนี้ ผู้เขียนก็จะขอเสนอการบีบอัดข้อมูลด้วยอัลกอริธึม LZSS ซึ่งเป็นอัลกอริธึมที่ได้รับการพัฒนาต่อเนื่องมาจากอัลกอริธึม LZ77 ทั้งนี้ก็เพื่อเป็นการขจัดปัญหาและเพิ่มประสิทธิภาพของอัลกอริธึม LZ77 เพราะหลังจากที่ได้มีการเผยแพร่อัลกอริธึม LZ77 ก็พบว่าในทางปฏิบัติโปรแกรมที่เขียนขึ้นตามแนวคิดของอัลกอริธึม LZ77 นั้นมีปัญหามากมายที่ไม่เหมาะกับการพัฒนาโปรแกรมในเชิงพาณิชย์ ด้วยเหตุนี้อัลกอริธึม LZSS จึงได้รับการพัฒนาจนเป็นที่ยอมรับจนถึงปัจจุบัน แต่เนื่องจากในปัจจุบันความสามารถของซีพียูนั้นได้รับการพัฒนาจนมีประสิทธิภาพดีกว่าในอดีตเป็นอย่างมาก ดังนั้นอัลกอริธึม LZ77 อาจจะสามารถเป็นที่ยอมรับได้ของโปรแกรมเมอร์บางท่าน

แนวคิดของการบีบอัดแบบ LZSS

ตามที่กล่าวมาแล้วในตอนที่ผ่านมาอัลกอริธึม LZ77 มีจุดอ่อนอยู่ที่ความเร็วในการประมวลผลของอัลกอริธึมนั้นมีความสัมพันธ์โดยตรงกับขนาดของ sliding window ดังนั้นถ้าหากขนาดของหน้าต่างดังกล่าวโตขึ้น 2 เท่า ก็จะส่งผลให้เวลาที่ใช้ในการประมวลผลมากขึ้นเกือบ 2 เท่าเช่นกัน แต่เป็นที่ทราบกันดีอยู่แล้วว่าถ้าหากขนาดของ sliding window โตขึ้น ก็จะช่วยลดจำนวนครั้งในการอ่านข้อมูลมายังบัฟเฟอร์ ซึ่งจะส่งผลให้อัลกอริธึมทำงานได้รวดเร็วขึ้นนั่นเอง ด้วยเหตุนี้จึงได้มีการ

ปรับปรุงอัลกอริธึม LZ77 ซึ่งในเวลาถัดมาก็ได้เกิดเป็นอัลกอริธึม LZSS ซึ่งอัลกอริธึมนี้ได้แก้ไขจุดอ่อน 2 ประการ ดังต่อไปนี้

1. อัลกอริธึม LZ77 นั้นมีการอ่านสายข้อมูลเข้ามายัง sliding window เพื่อทำการเข้ารหัสนั้น จะถูกจัดเก็บในรูปแบบของกลุ่มของข้อความอย่างต่อเนื่อง ส่วนอัลกอริธึม LZSS นั้นก็ยังคงหลักการนี้อยู่เพียงแต่ได้มีการเพิ่มเทคนิคของต้นไม้ไบนารี (binary tree) เช่นเดียวกับอัลกอริธึม huffman โดยการนำโครงสร้างต้นไม้ไบนารีมาใช้ในการจัดการสายข้อมูล โดยการทำการเรียงลำดับสายข้อมูลใน sliding window ลงในแต่ละปลายของต้นไม้ไบนารี ซึ่งด้วยวิธีนี้เวลาที่ใช้ในการประมวลผลหรือค้นหาสายสตริงที่ปลายที่ใกล้ที่สุดของต้นไม้ไบนารีนั้น จะมีค่าประมาณ log2(sliding window size) * (phrase string length) ซึ่งมีค่าน้อยกว่าเวลาที่ใช้สำหรับอัลกอริธึม LZ77 ที่มี sliding window ขนาดเดียวกัน ด้วยเหตุนี้ ขนาดของ sliding window สำหรับการบีบอัดข้อมูลด้วยอัลกอริธึม LZSS นั้นสามารถมีขนาดที่โตกว่าได้โดยที่เวลาที่ต้องใช้ในการประมวลผลนั้นเพิ่มขึ้นน้อยมาก

2. เนื่องจากอัลกอริธึม LZ77 นั้นจะบีบอัดข้อมูลโดยการ ส่งข้อมูลที่ถูกบีบอัดด้วยรูปแบบ <phrase offset><match length><character of encoded phrase> ถ้าหากสมมติว่าอัลกอริธึมไม่สามารถทำการบีบอัดสายสตริงใน sliding window นั้นก็หมายความว่า ไม่สามารถทำการจับคู่สายสตริงใน text window กับ look-ahead buffer ได้เลย ซึ่งในกรณีนี้อัลกอริธึม LZ77 ยังคงทำการเข้ารหัสในรูปแบบเดิมโดยการรายงานค่าในส่วนของ <match length> เท่ากับ 0 แต่สำหรับอัลกอริธึม LZSS นั้นได้มีการใช้ข้อมูลขนาด 1 บิต นำหน้าข้อมูลที่ถูกบีบอัด เพื่อเป็นการแสดงสถานะว่าข้อมูลที่ถูกบีบอัดที่ตามมาเป็นข้อมูลในรูปแบบ <phrase offset><match length> หรือตัวอักษรโดยตรง ด้วยเหตุนี้จึงทำให้ผลลัพธ์ของการบีบอัดข้อมูลดีขึ้นมากเลยทีเดียว

และเนื่องจากแนวคิดของอัลกอริธึม LZSS นี้มีพื้นฐานใกล้เคียงกับอัลกอริธึม LZ77 ดังนั้น เพื่อไม่ให้เป็นการเสียเวลาผู้เขียนก็ขอนำผู้อ่านเข้าสู่รายละเอียดของซอร์สโค้ดกันเลยดีกว่า เพราะจะช่วยให้ผู้อ่านสามารถเข้าใจเกี่ยวกับอัลกอริธึมได้รวดเร็วขึ้น

รายละเอียดของโปรแกรม LZSS

โปรแกรม LZSS เป็นโปรแกรมบีบอัดข้อมูลตามอัลกอริธึมแบบ dictionary-based ซึ่งในส่วนของตัวแปรชนิดครอบคลุมได้มีการประกาศค่าคงที่และตัวแปรชนิดครอบคลุมเอาไว้ ซึ่งมีรายละเอียดดังตารางที่ 1

ตารางที่ 1

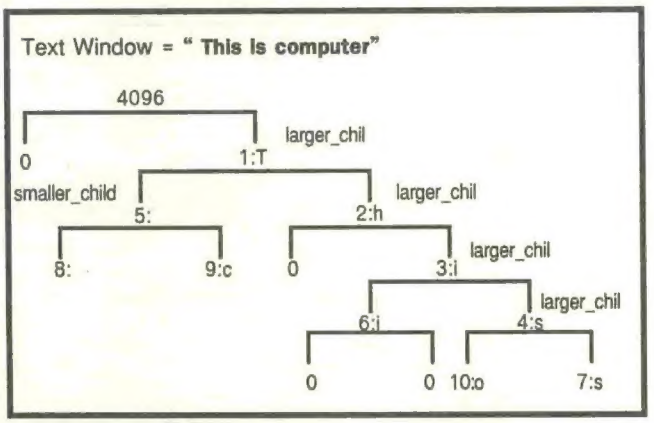
INDEX_BIT_COUNT	กำหนดจำนวนของบิตที่ใช้ในการกำหนดค่าของดัชนีที่บอกตำแหน่งในหน้าต่างข้อความ (text window) ซึ่งผู้อ่านสามารถที่จะแก้ไขได้ตามที่ต้องการ โดยค่าที่แก้ไขก็จะส่งผลโดยตรงต่อขนาดของหน้าต่างข้อความ ซึ่งในที่นี้กำหนดให้มีค่าเท่ากับ 12 บิต ซึ่งก็หมายความว่า ในโปรแกรมตัวอย่างมีการกำหนดให้หน้าต่างข้อความมีขนาดเท่ากับ 4,096 บิต
LENGTH_BIT_COUNT	กำหนดจำนวนของบิตที่ใช้ในการเข้ารหัสความยาวของข้อความที่ค้นพบ ซึ่งจะมีความสัมพันธ์โดยตรงกับขนาดของ look ahead buffer
WINDOW_SIZE	กำหนดขนาดของหน้าต่างข้อความ โดยในโปรแกรมตัวอย่างนี้ หน้าต่างข้อความจะมีขนาดเท่ากับ 4,096 บิต (1 << 12)
RAW_LOOK_AHEAD_SIZE	กำหนดขนาดของหน้าต่าง look-ahead ที่ใช้ในการเข้ารหัสความยาวของข้อความที่ค้นพบ ซึ่งในที่นี้จะมีค่าเท่ากับ 16 บิต (1 << 4)
BREAK_EVEN	ค่าตัวเลขที่ถูกใช้เป็นตัวกำหนดวิธีการเข้ารหัสข้อความให้เป็นแบบซนิด <index><length> หรือตัวอักษร โดยในที่กำหนดให้ค่าตัวเลข 17 เป็นค่าที่ใช้ในการเปรียบเทียบ โดยถ้าหากดูจากโปรแกรมตัวอย่างก็จะเห็นว่าถ้าหากข้อความที่เปรียบเทียบตรงกันเพียง 1 ตัวอักษรก็จะไม่เข้ารหัสตัวอักษร แต่ถ้าหากข้อความที่เปรียบเทียบมากกว่า 1 ตัวอักษร ก็จะเข้ารหัสข้อความดังกล่าวในรูปแบบ <index><length>
LOOK_AHEAD_SIZE	ค่าตัวเลขที่ถูกกำหนดให้ใช้ปรับค่าเริ่มต้นและค่าสิ้นสุดของ look ahead buffer ดังที่กล่าวมาแล้วข้างต้นว่าถ้าหากข้อความที่เปรียบเทียบตรงกันเพียง 1 ตัวอักษรก็จะไม่เข้ารหัสตัวอักษร ดังนั้นค่าตัวเลข 1 จึงไม่มีความจำเป็นที่จะต้องนำมาใช้บอกความยาวสตริงที่เข้ารหัส <length> ดังนั้นเพื่อเป็นการเพิ่มขนาดของlook ahead buffer จึงกำหนดให้ค่าตัวเลขบอกความยาว 0 ถึง 15 ในความเป็นจริงนั้นมีค่าเท่ากับความยาวของสตริงที่ถูกเข้ารหัส 2 ถึง 17 แทน
TREE_ROOT	ค่าตัวเลขกำหนดตำแหน่งของโหนดที่ชี้ไปยังตำแหน่งบนสุดหรือรากของแผนภูมิต้นไม้ ซึ่งในที่นี้หมายถึงโหนดที่ 4096 ซึ่งจะทำให้การเขียนโปรแกรมเพื่อค้นหาสตริงในแผนภูมิต้นไม้สามารถกระทำได้ง่ายขึ้น เพราะเราเพียงแต่เริ่มต้นที่กิ่งลูกของโหนดที่ 4096 เท่านั้น ก็เท่ากับว่าเราได้เริ่มต้นที่ตำแหน่งรากของแผนภูมิต้นไม้นั่นเอง ดังตัวอย่างในรูปที่ 1 แสดงให้เห็นตัวอย่างการจัดสตริงลงในแผนภูมิต้นไม้ →

ตารางที่ 1 (ต่อ)

`END_OF_STREAM`	ค่าตัวเลขเท่ากับ 0 ที่กำหนดจุดสิ้นสุดของสายสตริง
`UNUSED`	ค่าตัวเลขเท่ากับ 0 ที่กำหนดสถานะว่างของแต่ละโหนดและกิ่งของแผนภูมิต้นไม้
`MOD_WINDOW`	เป็นมาโครที่กำหนดให้ตำแหน่งสตริงที่อ่านยังคงอยู่ในพื้นที่ของหน้าต่างข้อความ
`window[ WINDOW_SIZE ]`	กำหนดขนาดของหน้าต่างข้อความสำหรับจัดเก็บสตริง 4,096 ตัวอักษรที่อ่านจากไฟล์ เพื่อทำการเข้ารหัส
`tree[ WINDOW_SIZE + 1 ]`	แผนภูมิต้นไม้สำหรับการจัดเก็บสตริงแบบเรียงลำดับ ทั้งนี้เพื่อให้ง่ายและรวดเร็วในการเข้าถึง และเป็นการขจัดปัญหาความล่าช้าในการประมวลผลอันเนื่องมาจากขนาดของหน้าต่างข้อความที่โตเกินไปอีกด้วย ซึ่งถูกกำหนดให้เป็นข้อมูลชนิดโครงสร้าง ที่ประกอบด้วย 3 สมาชิก เพื่อบอกตำแหน่งของโหนดแม่ โหนดลูกด้านน้อยกว่า และโหนดลูกด้านมากกว่าตามลำดับ ดังนี้

```
struct {
    int parent;
    int smaller_child;
    int larger_child;
} tree[ WINDOW_SIZE + 1 ];
```

สำหรับตัวอย่างการใช้ประโยชน์แผนภูมิต้นไม้ ผู้อ่านก็สามารถดูได้จากรูปที่ 1 ■



รูปที่ 1 แสดงให้เห็นตัวอย่างการจัดสตริงบางส่วนลงในแผนภูมิต้นไม้

ส่วนรูทีนหลักที่ควบคุมการเข้ารหัสข้อมูลนั้น เป็นรูทีนในไฟล์ LZSS.C ที่ชื่อ CompressFile ซึ่งการทำงานของรูทีนนี้ก็เริ่มต้นด้วยการอ่านสายสตริงเข้ามาเก็บเอาไว้ใน look ahead buffer และกำหนดให้ look_ahead_bytes มีค่าเท่ากับ 17 ซึ่งเท่ากับจำนวนตัวอักษรที่ถูกเข้ารหัสไปแล้ว (เพราะตัวอักษรชุดแรกจะไม่มีการเข้ารหัส) ด้วยโค้ดดังนี้

```
current_position = 1;
for ( i = 0 ; i < LOOK_AHEAD_SIZE ; i++ ){
    if ( ( c = getc( input ) ) == EOF )
        break;
    window[ current_position + i ] =
(unsigned char) c;
}
look_ahead_bytes = i;
```

จากนั้นก็เป็นการกำหนดค่าเริ่มต้นให้กับแผนภูมิต้นไม้ โดยการเรียกใช้รูทีน InitTree ซึ่งภายในรูทีนนี้ก็จะมีการกำหนดโหนดลูกด้านมากกว่าของโหนดรากมีค่าเท่ากับ 1 (โหนดปัจจุบัน) และโหนดแม่ของโหนดที่ 1 นี้ชี้ไปยังโหนดราก ส่วนโหนดลูกทั้งสองของโหนดที่ 1 มีค่าเท่ากับ 0 ดังนี้

```
tree[ TREE_ROOT ].larger_child = r;
tree[ r ].parent = TREE_ROOT;
tree[ r ].larger_child = UNUSED;
tree[ r ].smaller_child = UNUSED;
```

สำหรับส่วนสุดท้ายของการการกำหนดค่าเริ่มต้น ก็ต้องกำหนดให้ตัวแปร match_length และ match_position มีค่าเท่ากับ 0 ซึ่งถือว่าเป็นการเริ่มต้นเข้ารหัสข้อมูล โดยหลังจากนี้ก็จะเป็นโค้ดในวงลูป while เพื่อทำการอ่านสายสตริงเข้ามาเปรียบเทียบและเข้ารหัสต่อไป ซึ่งโค้ดในส่วนวงลูปนี้จะทำการเข้ารหัสข้อมูลไปเรื่อย ๆ จนกระทั่งไม่มีข้อมูลเหลือใน look ahead buffer หรือจนกระทั่ง look_ahead_bytes มีค่าเท่ากับ 0 นั่นเอง ซึ่งภายในวงลูปนี้เราสามารถแบ่งหน้าที่การทำงานออกเป็น 3 ส่วนหลัก ๆ ดังนี้

1. ทำการเข้ารหัสข้อมูลในส่วนปัจจุบัน ซึ่งสามารถเป็นได้ทั้งตัวอักษรหรือวลี (ส่วนของข้อความ) โดยเลือกวิธีการเข้ารหัสจากการเปรียบเทียบจากความยาวของวลีที่ค้นพบ (match_length) โดยถ้าหากมีค่าน้อยกว่าหรือเท่ากับค่าของ BREAK_EVEN ก็ไม่ต้องเข้ารหัสแต่จะจัดเก็บในรูปแบบดังนี้ <bit value 1><ascii> แต่ถ้าหากว่ามีความยาวมากกว่าค่าของ BREAK_EVEN ก็ให้ทำการเข้ารหัสวลีดังกล่าวด้วยรูปแบบ <bit value 0><index><length> ดังนี้

```
        if ( match_length > look_ahead_bytes )
            match_length = look_ahead_bytes;
        if ( match_length <= BREAK_EVEN ) {
            replace_count = 1;
            OutputBit( output, 1 );
            OutputBits( output,
                        (unsigned long) window
[ current_position ], 8 );
        } else {
            OutputBit( output, 0 );
            OutputBits( output,
                        (unsigned long)
match_position, INDEX_BIT_COUNT );
            OutputBits( output,
                        (unsigned long)
( match_length - ( BREAK_EVEN + 1 ) ),
                        LENGTH_BIT_COUNT );
            replace_count = match_length;
        }
```

2. อ่านสายสตริงเข้ามายัง look ahead buffer ในลักษณะของการอ่านเข้ามาต่อท้ายเรื่อย ๆ และทำการลบทิ้งข้อมูลในบัฟเฟอร์และแผนภูมิต้นไม้ที่เก่าที่สุดออกไปเท่ากับจำนวนตัวอักษรที่อ่านขึ้นมา

3. ทำการแทรกข้อมูลที่อ่านขึ้นมาใหม่ลงในแผนภูมิต้นไม้ ซึ่งโค้ดในขั้นตอนที่ 2 และ 3 นี้เป็นขบวนการที่ต้องทำไปพร้อม ๆ กัน โดยที่จำนวนตัวอักษรที่จะต้องอ่านขึ้นมาจากไฟล์เพื่อแทนที่ข้อมูลเดิมนั้น สามารถตรวจสอบได้จากค่าของ replace_count ส่วนจำนวนข้อมูลที่จะต้องทำการแทรกเข้าไปยังแผนภูมิต้นไม้นั้น ก็สามารถตรวจสอบได้จากค่าของตัวแปร look_ahead_bytes ดังนี้

```
    while ( look_ahead_bytes > 0 ) {
        for ( i = 0 ; i < replace_count;i++){
            DeleteString( MOD_WINDOW
(current_position + LOOK_AHEAD_SIZE ) );
            if ( ( c = getc( input )) == EOF )
                look_ahead_bytes--;
            else
                window[ MOD_WINDOW
(current_position + LOOK_AHEAD_SIZE ) ]
                        = (unsigned char) c;
          current_position = MOD_WINDOW
(current_position + 1 );
          if ( look_ahead_bytes )
              match_length = AddString
(current_position, &match_position );
        }
    };
```

และหลังจากที่ทำการเข้ารหัสข้อมูลจากไฟล์ทุก ๆ ตัวอักษรแล้ว เราก็ต้องทำการจัดเก็บค่าของ END_OF_STREAM ลงในไฟล์ข้อมูลเข้ารหัส เพื่อใช้เป็นค่าสำหรับบอกกับรูทีนที่ทำการถอดรหัสได้ทราบจุดสิ้นสุดของข้อมูลที่จะต้องทำการถอดรหัส ดังนี้

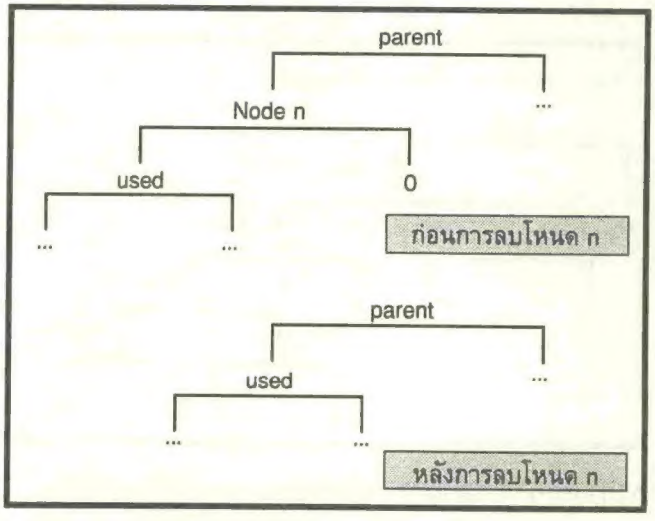
```
    OutputBit( output, 0 );
    OutputBits( output, (unsigned long)
END_OF_STREAM, INDEX_BIT_COUNT );
```

ซึ่งก็ถือว่าเป็นการสิ้นสุดขบวนการเข้ารหัสข้อมูลด้วยอัลกอริธึม LZSS แต่จากโค้ดข้างต้น ก็ยังมีอยู่ 2 รูทีนที่น่าสนใจ นั่นก็คือรูทีน DeleteString ที่ทำการลบทิ้งข้อมูลออกจากแผนภูมิต้นไม้และรูทีน AddString ที่ทำการเพิ่มข้อมูลเข้าไปยังแผนภูมิต้นไม้

โดยที่รูทีน DeleteString นี้จะถูกเรียกใช้ทุกครั้งที่มีการอ่านข้อมูลจากไฟล์มายัง look ahead buffer ทั้งนี้ก็เพื่อลบทิ้งข้อมูลเก่าที่สุดออกจากแผนภูมิต้นไม้ ซึ่งได้มีการเขียนโค้ดเพื่อดำเนินการดังกล่าวโดยอาศัยอัลกอริธึม Binary Tree Deletion

ซึ่งการทำงานภายในรูทีนนี้ก็เริ่มต้นด้วยการตรวจสอบว่าโหนดดังกล่าวอยู่ในแผนภูมิต้นไม้หรือไม่ จากนั้นจึงทำการลบทิ้งโหนด โดยมีวิธีในการลบทิ้ง 2 วิธีดังนี้

1. ถ้าหากมีโหนดลูกใดโหนดหนึ่งที่ไม่ถูกใช้ หรือมีค่าเท่ากับ 0 ก็ให้ทำการดึงโหนดแม่ดังกล่าวทิ้งออกจากแผนภูมิต้นไม้ และกำหนดหมายเลขโหนดดังกล่าวให้กับโหนดลูกที่ไม่ว่าง ซึ่งวิธีนี้ได้ถูกนำไปใช้ในรูทีน ContractNode ดังตัวอย่างในรูปที่ 2



รูปที่ 2 แสดงการลบโหนดออกจากแผนภูมิต้นไม้

2. กรณีที่โหนดลูกทั้งสองถูกใช้นั้น เราไม่สามารถที่จะลบทิ้งโหนดโดยวิธีการดึงออกจากแผนภูมิต้นไม้โดยตรง เพราะจะทำให้ข้อมูลในโหนดลูกตรงข้ามถูกลบทิ้งตามไปด้วย ดังนั้นในการลบทิ้งเราจะต้องเริ่มด้วยการค้นหาโหนดว่างทั้งก่อนและหลังโหนด n ซึ่งโค้ดที่ดำเนินการค้นหาโหนดว่างนี้จะอยู่ในรูทีน FindNextNode โดยจะถูกกำหนดให้เริ่มต้นจากซีกของโหนดที่มีค่าน้อยกว่าของโหนด n จากนั้นจึงค้นหาจากโหนดซีกที่มีค่ามากกว่า ดังนี้

```
    next = tree[ node ].smaller_child;
    while ( tree[ next ].larger_child !=
UNUSED )
        next = tree[ next ].larger_child;
    return( next );
```

จากนั้นก็ใช้หลักการของการเขียนโค้ดแบบเรียกตัวเอง (recursive) เพื่อทำการลบทิ้งโหนดว่างที่ค้นพบข้างต้นทั้งหมด แล้วจึงค่อยทำการแทนที่โหนดด้วยรูทีน ReplaceNode

```
    parent = tree[ old_node ].parent;
    if ( tree[ parent ].smaller_child ==
old_node )
        tree[ parent ].smaller_child =
new_node;
    else
        tree[ parent ].larger_child =
new_node;

      tree[ new_node ] = tree[ old_node ];
    tree[ tree[ new_node ].smaller_child ]
.parent = new_node;
    tree[ tree[ new_node ].larger_child ]
.parent = new_node;
    tree[ old_node ].parent = UNUSED;
```

ซึ่งโค้ดของรูทีน DeleteString ก็มีดังนี้

```
    if ( tree[ p ].parent == UNUSED )
        return;
    if ( tree[ p ].larger_child == UNUSED )
        ContractNode( p, tree[ p ]
.smaller_child );
    else if ( tree[ p ].smaller_child ==
UNUSED )
        ContractNode( p, tree[ p ]
.larger_child );
    else {
        replacement = FindNextNode( p );
        DeleteString( replacement );
        ReplaceNode( p, replacement );
    }
```

สำหรับในส่วนของรูทีน AddString นี้ ก็จะเริ่มต้นด้วยการค้นหาโหนดสำหรับการแทรกสตริงลงไปในแผนภูมิต้นไม้ ซึ่งก็จะใช้วิธีการเดินตามแต่ละโหนดของแผนภูมิต้นไม้นั่นเอง และทำการเปรียบเทียบค่าแอสกีระหว่าง new_node และ test_node โดยถ้าหาก new_node มีค่ามากกว่าค่าของ delta ก็จะเท่ากับ 1 แต่ถ้าหากมีค่าน้อยกว่าค่าของ delta ก็จะเท่ากับ -1 และจะมีค่าเท่ากับ 0 ถ้าหากค่าแอสกีระหว่าง new_node และ test_node เท่ากัน

```
        for ( i = 0 ; i < LOOK_AHEAD_SIZE ;
i++ ) {
            delta = window[ MOD_WINDOW
( new_node + i ) ] -
                    window[ MOD_WINDOW
( test_node + i ) ];
            if ( delta != 0 )
                break;
        }
```

เมื่อค้นพบโหนดที่มีค่าแตกต่างกันแล้ว ก็ต้องทำการเปรียบเทียบว่าความยาวของข้อมูลที่ได้มากกว่าความยาวของข้อมูลที่เคยค้นพบก่อนหน้านี้หรือไม่ โดยถ้าหากมีค่ามากกว่า ก็ให้ทำการแทนที่โหนดและจัดเก็บค่าความยาวของข้อมูลและตำแหน่งของโหนดใหม่ทันที

```
        if ( i >= match_length ) {
            match_length = i;
            *match_position = test_node;
            if ( match_length >=
LOOK_AHEAD_SIZE ) {
                ReplaceNode( test_node,
new_node );
                return( match_length );
            }
        }
```

และสำหรับโค้ดส่วนสุดท้ายของรูทีน AddString ก็จะเป็นการโยกย้ายข้อมูลลงในโหนดใหม่ที่ว่าง และกำหนดให้โหนดลูกของโหนดใหม่นี้มีสถานะว่าง และรูทีน AddString ก็จะมีการคืนค่ากลับเท่ากับความยาวของวลีที่เปรียบเทียบ ดังนี้

```
        if ( delta >= 0 )
            child = &tree[ test_node ]
.larger_child;
        else
            child = &tree[ test_node ]
.smaller_child;
        if ( *child == UNUSED ) {
            *child = new_node;
            tree[ new_node ].parent =
test_node;
            tree[ new_node ].larger_child =
UNUSED;
            tree[ new_node ].smaller_child =
UNUSED;
            return( match_length );
        }
       test_node = *child;
```

ข้อดีอีกประการหนึ่งของอัลกอริธึม LZSS ก็คือ การถอดรหัสข้อมูลสามารถกระทำได้เร็วกว่าการเข้ารหัสข้อมูลมากเพราะมีความซับซ้อนในการประมวลผลน้อยกว่า ดังนั้น จึงทำให้อัลกอริธึม LZSS เป็นที่นิยมกับการนำมาพัฒนาในเชิงธุรกิจ เพราะในการสร้างไฟล์ .EXE ที่สามารถแตกตัวเองได้ (self-extraction compressed file) นั้นก็จะทำให้ไฟล์ที่ได้มีขนาดเล็ก (โดยทั่วไปจะมีขนาดโตกว่าไฟล์บีบอัดทั่วไปประมาณ 1 ถึง 2 KB เท่านั้น) และสามารถประมวลผลได้อย่างรวดเร็ว

สำหรับรายละเอียดของโค้ดในส่วนของการถอดรหัสข้อมูล และซอร์สโค้ดทั้งหมดของการบีบอัดข้อมูลด้วยอัลกอริธึม LZSS นั้น ผู้เขียนก็จะขอกล่าวถึงในตอนถัดไปครับ พบกันใหม่ฉบับหน้าสวัสดีครับ